Mengen, Folgen

Liste, Keller, Schlange, Baum, Graph
charakteristischer Vektor, Hash-Tabelle

Feld, verkettete Liste, Adjazenzliste, Adjazenzmatrix...

Charakteristischer Vektor

Gegeben: feste, endliche Menge von Objekten

Gewünscht: direkter Zugriff auf ein Objekt

Lösung: Feld, bei dem jeder Position ein bestimmtes Objekt zugeordnet ist (eineindeutige Abbildung von Element auf Index).

Anwendungen

Skatkarten eines Spielers

S1: Kreuz:7,D; Pik:8,B,D; Herz:D,K; Karo:10,K;
S2: Kreuz:8,B,K,As; Pik:9,As; Herz:B,As; Karo:8,D;
S3: Kreuz:9,10; Pik:7,10; Herz:8,9,10; Karo:7,9,B;

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

t t t t t t t t t t

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

t t t t t t t t t t

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

t t t t t t t t t t

Kreuz Pik Herz Karo
7 8 9 10 b d k a 7 8 9 10 b d k as 7 8 9 10 b d k as 7 8 9 10 b d k as

Wenn nun S2 den Skat bekommt (Karo As, Herz 7), brauchen wir nur anzugeben:

```
skat[31]=true;
 skat[16]=true;
```

Eigenschaften

Einfügen und Löschen ist in O(1).

Schnittmenge a3[] aus a1[] und a2[] bilden:

```
for (i=1; a1.length()>i; i++) {
    a3[i]=a1[i] && a2[i];
}
```

Vereinigung:

```
for (i=1; a1.length()>i; i++) {
    a3[i]=a1[i] || a2[i];
}
```

Aufwand entspricht der Länge des Vektors (Ausgangsmenge), nicht der Menge der Einträge im Vektor.

Wenn der Vektor spärlich besetzt ist, ist der Aufwand zu groß!

Wir brauchen mehr Speicherplatz als notwendig.

Wörterbuch

Wenn wir nur Wörter mit einer Länge bis zu 10 Buchstaben hätten und keine Umlaute, dann gäbe es $26^{10} + 26^9 + 26^8 + ...26 = 10^{14}$ mögliche Wörter als Ausgangsmenge. Dabei sind aber nur ca. 10^8 Wörter in der Sprache realisiert.

Wörterbücher brauchen wir nicht nur für natürliche Sprachen, sondern auch für Programmiersprachen. Alle Namen (für Variable, Methoden, Klassen) müssen dem Übersetzer zur Verfügung stehen. Dafür gibt es eine **Symboltabelle**. In diese Tabelle werden alle Namen einmal eingetragen und bei dem Auftreten eines Namens im Programm wird geprüft, ob es den Namen schon gibt. Wieder können die Programmierer aus mindestens 10^{14} möglichen Namen auswählen, verwenden aber meist nur einen (neuen) Namen in 10 Zeilen Code! Für die meisten Programme reicht also eine Tabelle mit 300 Elementen.

Ein charakteristischer Vektor ist keine gute Idee für ein Wörterbuch oder eine ähnlich große Ausgangsmenge. Ein charakteristischer Vektor ist ungünstig, wenn wir die tatsächlich vorkommenden Einträge nicht vorher wissen.

Wir müssen den Vektor dichter packen.

Zerhacken von Feldern

Darstellung einer sehr großen Menge:

- jedes Element ist eindeutig bezeichnet (hat einen Schlüssel)
- Abbildung vom Element x auf eine Speicheradresse:
  $f(x) \to \mathbb{N}$

Die Funktion zerhackt den Speicher. Deshalb heißt sie **Hash-Funktion**.

Eine Hash-Funktion bildet mögliche Elemente einer Menge auf eine feste Anzahl von Adressen ab.

Finden der Hash-Funktion

- Ist der charakteristische Vektor hierfür eine gute Idee? Schlüssel ist der charakteristische Vektor und die Hash-Funktion ist seine Interpretation als Zahl.
  Diese Funktion ist eineindeutig!
  Also verschwendet sie Speicherplatz.

- Perfekt ist die Funktion, wenn sie eindeutig die Adresse für ein Element angibt, aber nicht für jeden Speicherplatz das Element eindeutig angeben kann.
  Es soll nur 1 Element auf 1 Adresse abgebildet werden und
  es soll so wenig als möglich Adressen geben.

Menge: {braun, rot, blau, violett, türkis}

$f(x)$ sei Wortlänge(x) - 3

Speicher:

0: rot, 1: blau, 2: braun, 3: türkis, 4: violett

Hash-Funktion für Wörter

Ein Wort ist eine Folge von Buchstaben

$w = c_0, c_1, ...c_{n-1}$.

Jedem Buchstaben ist ein Zahlenwert zugeordnet. Z.B.: A - 65 ... Z - 90, a - 97 ... z - 122;

Dann ist die Summe dieser Zahlenwerte $modulo$ der Anzahl der Adressen B (d.h. der Rest, der übrig bleibt, wenn die Summe durch B geteilt wird) die Adresse. Sie liegt gewiß im Intervall von 0 bis B.

$f(w) = \left(\sum_{i=0}^{n-1} c_i\right)$ modulo B

Hut: 7 + 117 + 116 = 305 ergibt modulo 3 Adresse 2

Mantel: 77 + 97 + 110 + 116 + 101 + 108 = 609 ergibt modulo 3 Adresse 0

Stock: 83 + 116 + 111 + 99 + 107 = 516 ergibt modulo 3 Adresse 0

Offenes Hashing

Werden mehrere Elemente auf dieselbe Adresse abgebildet, so heißt dies **Kollision**.

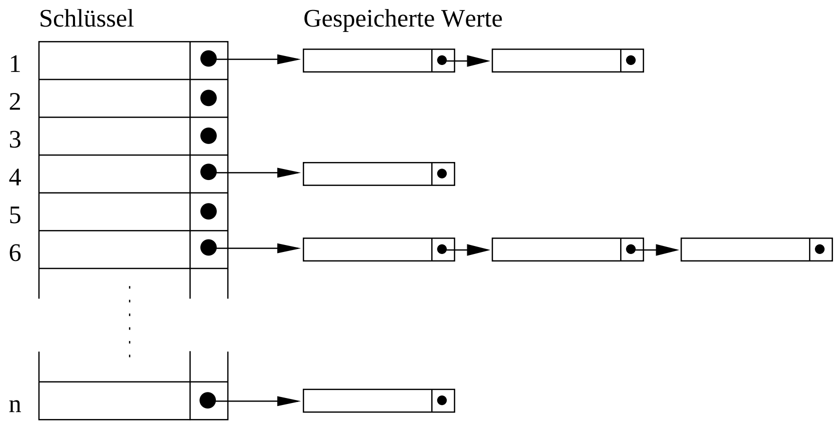
Die ursprüngliche Adresse ist der Anfang einer verketteten Liste. Die verkettete Liste wird um jedes Element, das auf diese Adresse abgebildet wird, verlängert.

Adresse 0 zeigt auf die Liste Mantel, Stock

(engl.: bucket, d.h. Eimer)

Die Hash-Funktion soll die Elemente möglichst gleichmäßig über die Adressen verteilen, d.h. es sollen möglichst wenig Kollisionen bestehen, denn eine Liste zu durchsuchen, dauert länger als mit der Hash-Funktion direkt zuzugreifen.

Hash-Tabelle – offenes Hashing



Die zu speichernde Menge darf nun beliebig wachsen. Es verändert sich der Lastfaktor, d.h. wieviele Speicherzellen benutzt werden.

Der **Lastfaktor** einer Hash-Tabelle ist bei der Anzahl B von Adressen und der Kardinalität M der darzustellenden Menge das Verhältnis M/B.

Wenn M/B etwa 1 ist, ist der Aufwand, ein Element einzufügen oder zu finden $O(1)$.

Wir sehen vielleicht anfangs für die erwartete Kardinalität der Menge $M = 1.000.000$ einen Lastfaktor von 1 vor. Dann wächst die Menge immer mehr. Bei $E = 1.080.000$ Einträgen haben wir einen Lastfaktor von 1,35. Wir beschließen, eine neue Hash-Tabelle mit $M' = B = 3 \cdot E =$3.240.000 anzulegen, sehen also wieder einen Lastfaktor von 1 vor. Das Anlegen der neuen Tabelle erfordert $O(M')$ Aufwand. Aber das Auffinden eines Elements in der neuen Tabelle ist jetzt wieder nahe $O(1)$.

Hashtable in JAVA

Klasse unter **Dictionary** mit den Eigenschaften

- int capacity – für B und
- $float$ loadFactor – für M/B.

Der Lastfaktor soll zwischen 0 und 1 liegen. Wenn die Anzahl der Einträge E größer wird als Lastfaktor mal Anzahl der Adressen, wird die Anzahl der Adressen erhöht und neu abgebildet (re-hash).

$$E \leq M/B \cdot B = E \leq M$$

Wenn der Lastfaktor nicht angegeben ist, wird 0,75 angenommen. Die Tabellengröße ist initial 101. Beim re-hash wird $3 \cdot E$ als neue Tabellengröße gewählt.

Eintragen

Anlegen einer Hashtabelle und eintragen von Elementen:

Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));

put(**Object** Schlüssel, **Object** Element)

Der Schlüssel ist ein eindeutiger Bezeichner x, die Hash-Funktion liefert den Index der Hash-Tabelle tab[i]: $f(x) = i$.

Die Hash-Funktion ist in der virtuellen Maschine für jedes Objekt bei der Klasse **Object** angegeben und wird nicht im JAVA-Programm deklariert, d.h. Programmiererinnen könnnen sie nicht bestimmen.

Die Methode **hashCode**() von **Hashtable** gibt lediglich den int Funktionswert der Hash-Funktion zurück.

Die Hash-Tabelle ist ein Feld von Einträgen. Ein Eintrag ist eine private Klasse mit den Eigenschaften

- int hash – für $f(x) = i$ und
- $Object$ key – für den eindeutigen Bezeichner, z.B. "one"
- $Object$ value – für den eigentlichen Eintrag, z.B. Integer(1)
- $Entry$ next – für den Listennachfolger.

Bei einem neuen Eintrag werden folgende Schritte gemacht:

- Position in der Hash-Tabelle finden:

  int hash= key.hashCode();

  Dieser Wert wird noch etwas verändert (modulo tab.length) und ist dann der Index.

- Feststellen, ob es den Schlüssel nicht schon gibt: Solange tab[index] nicht null ist, wird zum Listennachfolger gegangen. Wenn der Schlüssel schon vergeben ist, wird der Wert zu diesem Schlüssel geändert. Es wird kein Schlüssel doppelt eingetragen! Sonst wird der neue Eintrag hinten in die Liste gehängt.

- Gegebenenfalls re-hash

- eintragen: tab[index]= e;

Aufwand

Der ungünstigste Aufwand ist in O(M), wenn alle Elemente denselben Hash-Wert haben. Wir durchsuchen eine M lange Liste!

Wenn die Hash-Funktion gleichmäßig verteilt, erwarten wir aber O(1 + Lastfaktor). Die Wahrscheinlichkeit für $f(x) = i$ ist 1/B. Die Wahrscheinlichkeit, daß ein Element den Index i hat ist 1/M. Wir haben vermutlich M/B Listen. Der tatsächliche Aufwand hängt also entscheidend von dem Lastfaktor ab!

Zugriff

Zugriff auf ein Element der Tabelle:

Integer n = (Integer)numbers.get("two");
if (n != null)
System.out.println("two = "+n);

get(**Object** Schlüssel)

size() gibt an, wieviele Schlüssel vergeben wurden, d.h. wieviele Listen (Eimer) nicht leer sind, also E.

```
/*
* HashBeispiel
*
* Ein Beispielprogramm, um den Gebrauch einer Hashtable
* zu demonstrieren.
*
*/

import AlgoTools.IO;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;

class HashBeispiel {

    public static void main (String[] argv) {
        Hashtable lexikon;                        // Unser Lexikon
        Object[] words;           // Ein Feld mit allen Begriffen
        Random zufall = new Random ();            // Ein Zufalls-
                                                  // zahlengenerator
        int i;                                    // Eine Zufallszahl
        String begriff, geraten, nochmal;

        lexikon = new Hashtable ();               // Erzeugen des
                                                  // Lexikons
```

```
                                    // Ein paar Begriffe
                                                   //
lexikon.put ("Auto", "Hat 4 Raeder, produziert
                                          Abgas");
lexikon.put ("Fahrrad", "Hat 2 Raeder, produziert
                                        Schweiss");
lexikon.put ("Flugzeug", "Fliegt schnell und
                            manchmal zuverlaessig.");
lexikon.put ("Rakete", "Fliegt schnell und
                          transportiert Satelliten");
lexikon.put ("Skates", "Hat 8 Raeder, produziert
                                          Spass");
lexikon.put ("Vogel", "Fliegt langsam und legt Eier");

IO.println ("Hallo ! Willkommen beim Begrifferaten
                                               !");
IO.println ("Das Lexikon enthaelt "+ lexikon.size ()
                                    + "Begriffe.");

words = lexikon.keySet ().toArray();     // Menge der
                                  // Woerter bestimmen
```

```
    // Hauptschleife. Gibt Bedeutung und fragt Benutzer
                                            // nach Begriff
    do {
                                    // Einen Begriff auswaehlen
        i = zufall.nextInt (lexikon.size ());
        begriff = words[i].toString();

                                                // Abfragen
        IO.println ();
        IO.println (lexikon.get (begriff).toString());
        geraten = IO.readString ("Was ist das ? ");

        if (geraten.equals (begriff))
        IO.println ("Hurra ! Richtig !!");
        else
        IO.println ("Leider falsch. Die richtige Antwort
                                        lautet: "+ begriff);

        nochmal = IO.readString ("Nocheinmal
                                            (ja/nein) ? ");
    } while (nochmal.equals ("ja"));

    IO.println ("Auf Wiedersehen beim Begrifferaten.");
  }
}
```